Programmation avancée

ENSIIE

Semestre 2 — 2013–14

Objectifs du cours

- Représentation de données non linéaires en C
- Données persistantes/mutables
- Gestion de la mémoire
- Complexité
- Compilation séparée et modularité
- Structures de données avancées (tables de hash, . . .)

Représentation de structures arborescentes

Arbres : rappels

- Permettent de stocker les données de manière compacte. La *hauteur* d'un arbre est la plus grande longueur d'un chemin de la racine à une feuille.
- Les points d'un arbre sont des *noeuds* qui ont des *fils*, ou des *feuilles*. Tout point d'un arbre qui n'est pas la racine possède un *père*.
- Un *arbre binaire* de hauteur h permet de stocker dans ses *etiquettes* et ses feuilles $2^h - 1$ valeurs.

Structures d'arbres

Données très utilisées en informatique

- HTML, XML : les *feuilles* sont du texte, les *noeuds* sont les balises. La *racine* est la balise `HTML`.
- La syntaxe abstraite d'un langage de programmation. Le texte source d'un programme est transformé en un arbre de syntaxe abstraite (AST).
- Dans un système de fichiers, les points sont des dossiers ou des fichiers.
- ...

Première méthode : comme une structure algébrique

Méthode générique qui permet d'encoder les types algébriques (types somme) à la OCaml
Exemple :

```
type t =
  A
| B of int
| C of t * char
```

Structure algébrique

```
enum cas_t {A, B, C};

union union_t {
  /* cas A : pas de contenu */
  /* cas B */   int b;
  /* cas C */   struct donnee_C * c;   };

struct donnee_t {
  enum cas_t genre;
  union union_t contenu; };

typedef struct donnee_t t;

struct donnee_C {
  t fst;
  char snd;    };
```

Constructeurs

```
t a () {
  t res;
  res.genre = A;
  return res;
}
```

```
t b (int i) {
  t res;
  res.genre = B;
  res.contenu.b = i;
  return res;
}
```

```
t c (t i, char j) {
  t res;
  res.genre = C;
  res.contenu.c = malloc(sizeof(struct donnee_C));
  res.contenu.c->fst = i;
  res.contenu.c->snd = j;
  return res;
}
```

Exercice

Définir les arbres binaires de cette façon.

Deuxième méthode : forêt de pointeurs

On accède aux sous-arbres à l'aide de pointeurs.
Le pointeur NULL représente l'arbre vide : si un des fils d'un nœud est NULL, alors en fait ce fils n'existe pas.

```
typedef struct donnee_arbre* arbre_binaire;

struct donnee_arbre {
  int etiquette;
  arbre_binaire fils_gauche;
  arbre_binaire fils_droit;
}
```

Constructeurs

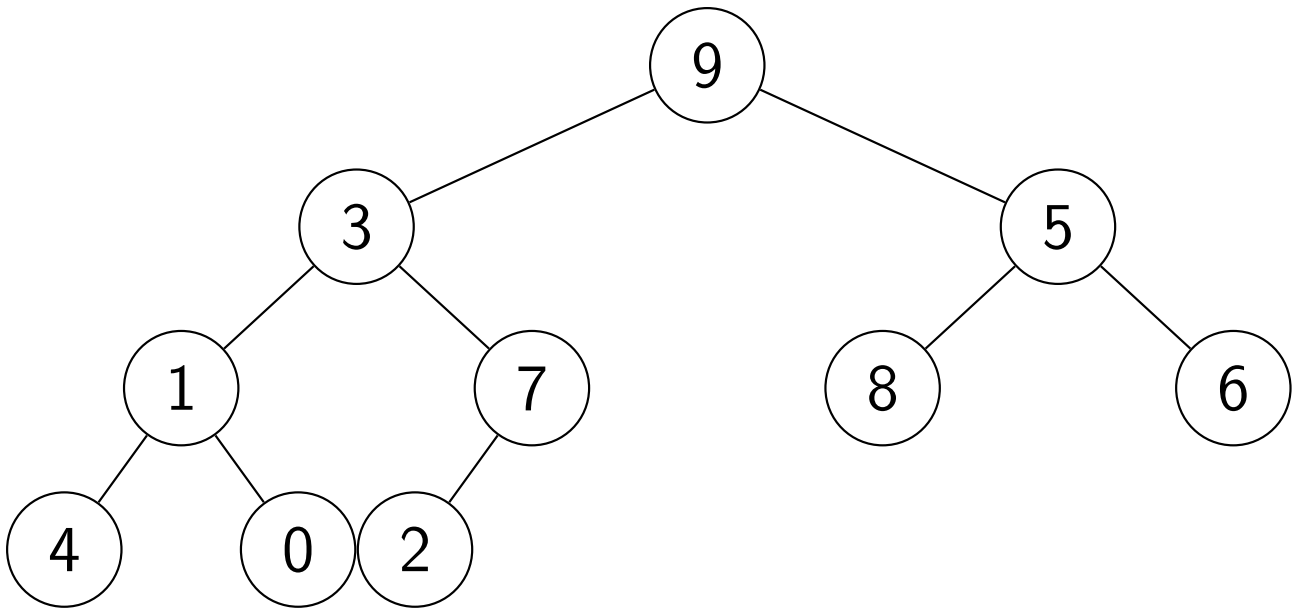
```
arbre_binaire empty() {
  return NULL;
}

arbre_binaire node(arbre_binaire fg,
                   int etiquette,
                   arbre_binaire fd) {
  arbre_binaire res =
    malloc(sizeof(struct donnee_arbre));
  res->fils_gauche = fg;
  res->fils_droit = fd;
  res->etiquette = etiquette;
  return res;
}
```

Troisième méthode : tableau

Les arbres binaires peuvent se coder en un tableau t : les valeurs $t[2^{i-1}-1]$ à $t[2^i-2]$ correspondent à celle à la hauteur i.

Exemple :



est représenté par

9	3	5	1	7	8	6	4	0	2

Avantages

Représentation mémoire compacte et rapide

Accès en temps constant

- au fils gauche ($n \mapsto 2 \times n + 1$)
- au fils droit ($n \mapsto 2 \times n + 2$)
- au père ($n \mapsto (n - 1)/2$)
- à l'étiquette ($n \mapsto t[n]$)

Utile uniquement si l'arbre est presque complet :

- tous les niveaux sont remplis
- sauf le dernier où les nœuds sont le plus à gauche

Sinon présence de "trous" dans le tableau

Persistance/mutabilité

En programmation fonctionnelle, les données sont persistantes :

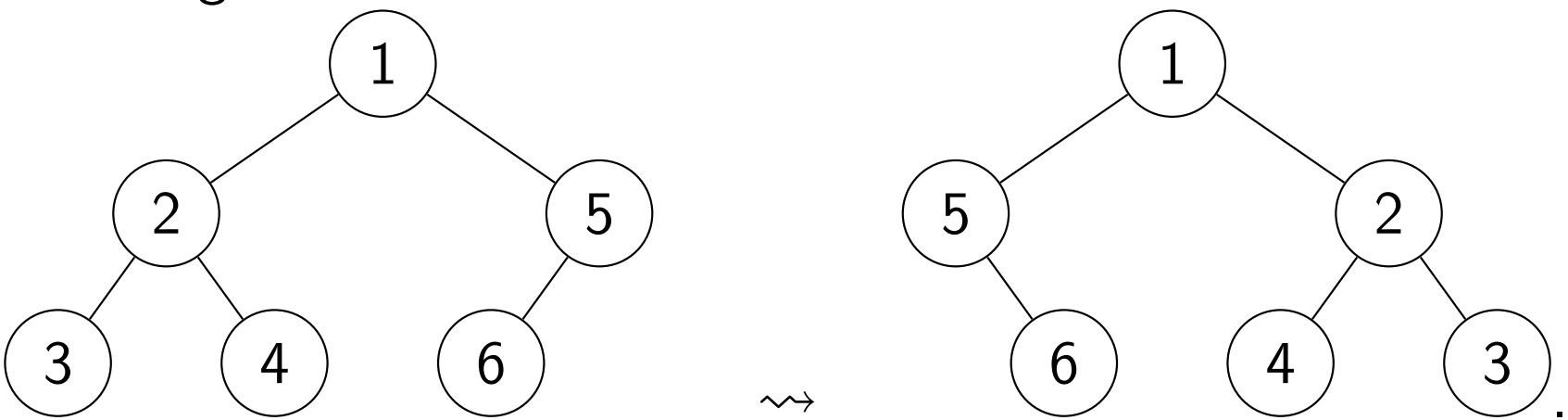
- une fois définies, elles ne peuvent pas être modifiées

Au contraire, en programmation impérative, les données sont mutables. Pour un algorithme donné, on trouvera donc deux versions :

- version destructive : les changements se font sur place, la structure d'origine n'est plus accessible
- version non-destructive : on crée une nouvelle structure pour faire le changement, l'ancienne est toujours là

Exemple : fonction miroir

On écrit la fonction qui prend un arbre en argument et renvoie son image dans un miroir :



Version destructive

```
arbre_binaire miroir(arbre_binaire a) {
  arbre_binaire tmp;
  if (a) {
    tmp = a->fg;
    a->fg = miroir(a->fd);
    a->fd = miroir(tmp);
    return a;
  }
  else return NULL;
}
```

Version non destructive

```
arbre_binaire miroir(arbre_binaire a) {
  if (a)
    return (node(miroir(a->fd),
                 a->etiquette,
                 miroir(a->fd)));
  else
    return NULL;
}
```